ANNOUNCEMENTS

Project 2a: Graded – see Learn@UW; contact your TA if questions
Part 2b will be graded end of next week

• If you have an academic conflict, fill out form.
• Covers all of Concurrency Piece (lecture and book) + I/O and RAID
  • Light on chapter 29, nothing from chapter 33; chapters 36, 37, 38
  • Few review questions from Virtualization Piece
• Look at homework simulators
• Questions from Project 2
• See previous practice exam

Project 3: Extension til Friday 11/04 9:00 pm
• Use name “stats_” and not “stat_”

Today’s Reading: Chapter 32

COND VAR: REVIEW

The scheduler runs each thread in the system such that each line of the given C-language code executes in one scheduler tick, or interval. For example, if there are two threads in the system, T and S, we tell you which thread was scheduled in each tick by showing you either a “T” or a “S” to designate that one line of C-code was scheduled for the corresponding thread; for example, TTTTS means that 3 lines of C code were run from thread T followed by 2 lines from thread S.

Some lines of C code require special rules, as follows.

- Assume each line of a while() loop or an if() statement requires one scheduler tick. Assume jumping to the correct code does not take an additional tick (e.g., jumping either inside or outside the while loop or back to the while condition does not take an extra tick; jumping to the then or the else branch of an if statement does not take an extra tick).

- Assume function calls whose internals are not shown and that do not require synchronization, such as malloc(), free(), fprintf(), printf(), printf(), and malloc(), require one scheduling tick.

- Function calls that may need to wait for another thread to do something (e.g., mutex_lock() and cond_wait()) may consume an arbitrary number of scheduling ticks and are treated as follows.

For mutex_lock(), assume that the function call to mutex_lock() requires one scheduling interval if the lock is available. If the lock is not available, assume the call spin-waits until the lock is available (e.g., you may see a long instruction stream TTTTTT that causes no progress for this thread). Once the lock is available, the next scheduling of the acquiring thread causes that thread to obtain the lock (e.g., after a thread S releases the lock, the next scheduling of the waiting thread T will complete mutex_lock(); note that T does not need to be scheduled for one tick with the lock released for mutex_lock() to complete).

The rules for cond_wait() and sema.wait() are similar. When a thread calls one of these versions of wait(), if the work has not yet been done to complete the wait, then no matter how long the scheduler runs this thread (e.g., TTTT), this thread will remain waiting in the wait() routine. After another thread runs and does the work necessary for the wait() routine to complete, then the next scheduling of thread T will cause the wait() (s) to complete; again, note that T does need to be scheduled for one tick with the work completed for wait() to complete.

Today’s Reading: Chapter 32
After the instruction stream “P” (i.e., after scheduler runs one line from parent), which line of the parent’s will execute when it is scheduled again? p2. **P will have acquired lock and finished mutex_lock().**

Assume the scheduler continues on with “C” (the full instruction stream is PC). Which line will child execute when it is scheduled again?

C1. **C must wait to acquire the lock since it is currently held by P.**

After PPP (full is PCPPP), which line for parent next?

p3. Since done = 0, P will execute p2 and p3; it is stuck in p3 until signaled.

After CCC (full is PCPPPPCCC), which line for child next?

c4. C finishes c1, c2, c3. Next time it executes c4. CV signaled but mutex still locked

After PP (full is PCPPPCCPPP), which line for parent next?

p3. P cannot return from cond_wait() until it acquires lock held by C; P stays in p3

After CC (full is PCPPPPCCPPCCC), which line for child next?

**Beyond. C executes c4 and then code beyond c4.**

After PPP (full PCPPPPCCPPCCPPP), which line for parent next?

**Beyond. P finishes p3, rechecks p2, then p4. Next time beyond p4.**

---

**CONCURRENCY BUGS**

**Questions answered in this lecture:**

- What type of concurrency bugs occur?
- How to fix **atomicity bugs** (with locks)?
- How to fix **ordering bugs** (with condition variables)?
- How does **deadlock** occur?
- How to prevent deadlock (with waitfree algorithms, grab all locks atomically, trylocks, and ordering across locks)?
CONCURRENCY IN MEDICINE: THERAC-25 (1980’S)

“The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a race condition.”

“…in three cases, the injured patients later died.”


CONCURRENCY STUDY FROM 2008

Lu etal. Study:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

MySQL: Atomicity, Order, or Deadlock?

Thread 1:
if (thd->proc_info) {

...

fputs(thd->proc_info, ...);

...

}

Thread 2:

thd->proc_info = NULL;

What’s wrong?
Test (thd->proc_info != NULL) and set (writing to thd->proc_info) should be atomic

MySQL: Fix Atomicity Bugs with Locks

Thread 1:

pthread_mutex_lock(&lock);
if (thd->proc_info) {

...

fputs(thd->proc_info, ...);

...

}

pthread_mutex_unlock(&lock);

Thread 2:

pthread_mutex_lock(&lock);

thd->proc_info = NULL;

pthread_mutex_unlock(&lock);
MOZILLA: ATOMICITY, ORDERING, OR DEADLOCK

Thread 1:
```c
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}
```

Thread 2:
```c
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

What's wrong?

Thread 1 sets value of mThread needed by Thread2
How to ensure reading mThread happens after mThread initialization?

---

FIX ORDERING BUGS WITH CONDITION VARIABLES

Thread 1:
```c
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}
```

Thread 2:
```c
void mMain(...) {
    Mutex_lock(&mtLock);
    while (mtInit == 0)
        Cond_wait(&mtCond, &mtLock);
    Mutex_unlock(&mtLock);
    mState = mThread->State;
    ...
}
```
**CONCURRENCY STUDY FROM 2008**

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.


---

**DEADLOCK**

Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does.

“Cooler” name: **deadly embrace** (Dijkstra)
Both cars arrive at same time
Is this deadlocked?
who goes?

B

A

STOP

STOP

STOP

STOP
4 cars arrive at same time
Is this deadlocked?

who goes?
4 cars move forward same time
Is this deadlocked?

Deadlock!
CODE EXAMPLE

Thread 1:
lock(&A);
lock(&B);

Thread 2:
lock(&B);
lock(&A);

Can deadlock happen with these two threads?

CIRCULAR DEPENDENCY
**FIX DEADLOCKED CODE**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(&amp;A);</td>
<td>lock(&amp;B);</td>
</tr>
<tr>
<td>lock(&amp;B);</td>
<td>lock(&amp;A);</td>
</tr>
</tbody>
</table>

How would you fix this code?

**NON-CIRCULAR DEPENDENCY (FINE)**

![Diagram showing non-circular dependency](Image)
**WHAT'S WRONG?**

```c
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = Malloc(sizeof(*rv));
    Mutex_lock(&s1->lock);
    Mutex_lock(&s2->lock);
    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    }
    Mutex_unlock(&s2->lock);
    Mutex_unlock(&s1->lock);
}
```

**ENCAPSULATION**

Modularity can make it harder to see deadlocks

Thread 1:
```
rv = set_intersection(setA, setB);
```

Thread 2:
```
rv = set_intersection(setB, setA);
```

Solution?
```
if (m1 > m2) {
    // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

Any other problems?
Code assumes m1 != m2 (not same lock)
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating any one condition

Mutual Exclusion

Def:

Threads claim exclusive control of resources that they require (e.g., thread grabs a lock)
WAIT-FREE ALGORITHMS

Strategy: Eliminate locks!

- Try to replace locks with atomic primitive:

```c
int CompAndSwap(int *addr, int expected, int new)
Returns 0: fail, 1: success
```

```c
void add (int *val, int amt) {
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}
```

```c
void add (int *val, int amt) {
    do {
        int old = *value;
    } while(!CompAndSwap(val, old, old+amt));
}
```

WAIT-FREE ALGORITHMS: LINKED LIST INSERT

Strategy: Eliminate locks!

```c
int CompAndSwap(int *addr, int expected, int new)
Returns 0: fail, 1: success
```

```c
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
    do {
        n->next = head;
    } while (!CompAndSwap(&head, n->next, n));
}
```
**DEADLOCK THEORY**

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating any one condition

---

**BREAK**

- What was your best halloween costume?
- Do you have a halloween costume yet for this year?
**HOLD-AND-WAIT**

**Def:**

Threads hold resources allocated to them (e.g., locks they have already acquired) while waiting for additional resources (e.g., locks they wish to acquire).

**ELIMINATE HOLD-AND-WAIT**

**Strategy:** Acquire all locks atomically **once**

- Can release locks over time, but cannot acquire again until all have been released

**How to do this?** Use a meta lock, like this:

```c
lock(&meta);
lock(&L1);
lock(&L2);
...
unlock(&meta);
```

// Critical section code

```c
unlock(...);
```

**Disadvantages?**

- Must know ahead of time which locks will be needed
- Must be conservative (acquire any lock possibly needed)
- Degenerates to just having one big lock
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating any one condition

No Preemption

Def:

Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
**SUPPORT PREEMPTION**

Strategy: if thread can't get what it wants, release what it holds

```
top:
    lock(A);
    if (trylock(B) == -1) {
        unlock(A);
        goto top;
    }
```

Disadvantages?

Livelock:
no processes make progress, but the state
of involved processes constantly changes
Classic solution: Exponential back-off

**DEADLOCK THEORY**

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold and wait
- no preemption
- circular wait

Eliminate deadlock by eliminating any one condition
CIRCULAR WAIT

Def:

There exists a circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.

ELIMINATING CIRCULAR WAIT

Strategy:

- decide which locks should be acquired before others
- if A before B, never acquire A if B is already held!
- document this and write code accordingly

Works well if system has distinct layers
LOCK ORDERING IN LINUX

In linux-3.2.51/include/linux/fs.h

/* inode->i_mutex nesting subclasses for the lock
 * validator:
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: quota file
 * The locking order between these classes is
 * parent -> child -> normal -> xattr -> quota
 */

HOMEWORK

[HW-Threads-RealDeadlock]

Look at code and run:

  vector-deadlock.c
  vector-global-order.c
  vector-avoid-hold-and-wait.c
  vector-nolock.c
  vector-try-wait.c
SUMMARY

When in doubt about correctness, better to limit concurrency (i.e., add unnecessary lock)

Concurrency is hard, encapsulation makes it harder!

Have a strategy to avoid deadlock and stick to it

Choosing a lock order is probably most practical